Gli algoritmi euristici

In questa pagina, passiamo in rassegna alcuni algoritmi euristici per il *TSP*. Ci concentriamo su questo problema perché è il più semplice problema di *routing* e *scheduling* e permette quindi di sottolineare più chiaramente i concetti fondamentali che sottendono la concezione di algoritmi efficaci per tali problemi.

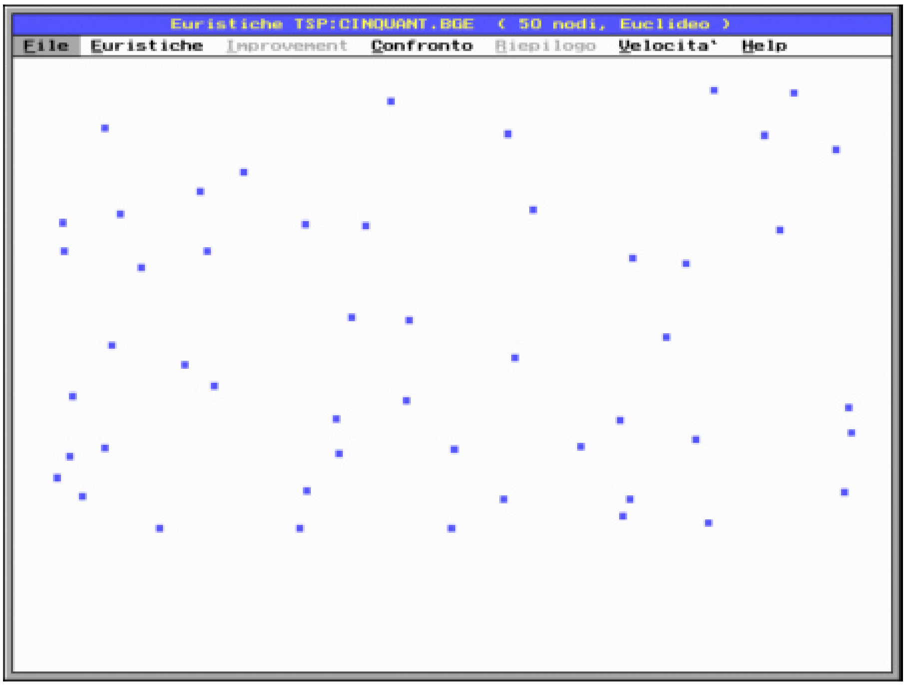
Esistono moltissime famiglie di algoritmi euristici, ispirate ad approcci diversi, più o meno sofisticati. Nel seguito, pur accennando a tutti gli approcci principali, ci concentreremo sugli *algoritmi costruttivi*, cioè quelli nei quali la soluzione viene ottenuta a partire da zero, attraverso una sequenza di soluzioni parziali che vengono via via completate. Nel caso del *TSP*, ciò avviene aggiungendo ad ogni passo un nuovo nodo, fino a ottenere una soluzione ammissibile, cioè un ciclo che passi per tutti i nodi del grafo.

Ci concentriamo su questi algoritmi perché sono i più semplici e accessibili a chi non sia esperto del campo, perché sono già sufficienti a sottolineare alcuni concetti chiave e perché gli algoritmi disponibili nel software *DaR*, pur essendo più sofisticati, ricadono nella stessa famiglia.

Vedremo così:

- l'algoritmo *Nearest neighbour* (ovvero, dall'inglese, *Vicino più prossimo*)
- l'algoritmo *Nearest insertion* (ovvero, dall'inglese, *Inserzione del nodo più prossimo*)
- l'algoritmo *Farthest insertion* (ovvero, dall'inglese, *Inserzione del nodo più lontano*)
- l'algoritmo *Sweep* (ovvero, dall'inglese, *Scorrimento circolare*)

Questi algoritmi non vanno certo intesi come un elenco esaustivo, ma solo come esempi di algoritmi costruttivi. Ne seguiremo l'esecuzione passo per passo, riferendoci al grafo riportato nella figura seguente.



Un grafo di esempio da 50 nodi

Nella figura sono rappresentati i nodi e non gli archi, perché il grafo va inteso come completo (cioè ogni coppia di nodi è legata da un arco) e il costo di ogni arco è dato dalla *distanza euclidea*, cioè la distanza in linea d'aria, fra i due nodi estremi.

Quindi, passeremo a considerare le altre famiglie di algoritmi:

- gli algoritmi di *ricerca locale classica*
- le *metaeuristiche di ricerca locale*, come il Tabu Search e il Simulated Annealing
- gli *algoritmi genetici*
- l'*Ant System*

Top

Algoritmo *Sweep*

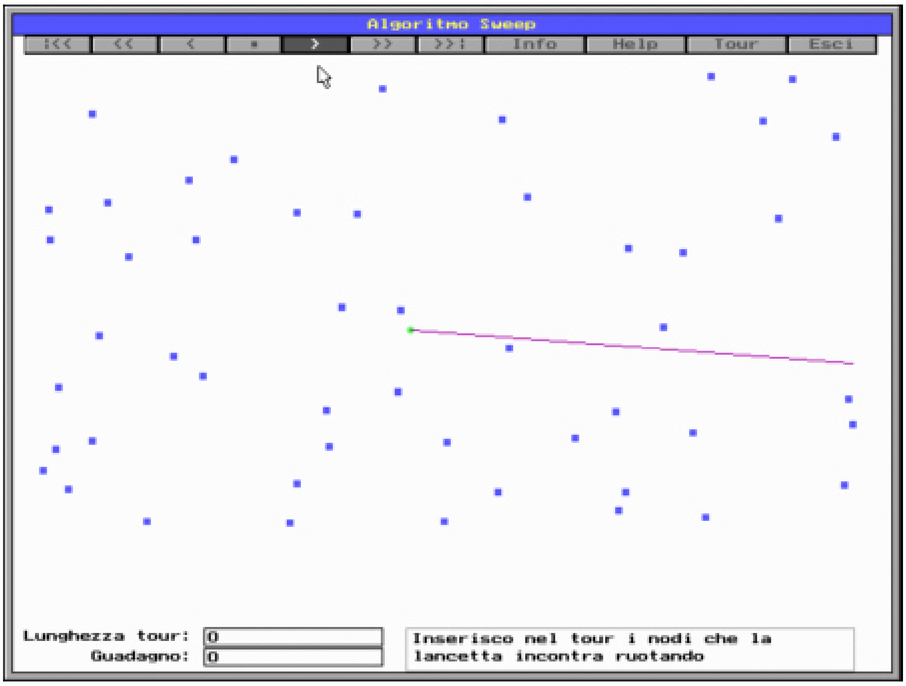
L'algoritmo *Sweep* si basa su un'idea molto intuitiva della visita di luoghi distribuiti nello spazio. L'idea è di dividere in settori l'area su cui risiedono i clienti e visitarli nell'ordine.

Più precisamente, si fissa un punto centrale nell'area e da esso si trae un raggio che scorre l'area stessa, in verso orario o antiorario. Il percorso del commesso viaggiatore viene costruito collegando via via, nell'ordine, i nodi toccati dal raggio.

Questo algoritmo ha un vantaggio teorico, dato che le sue soluzioni godono di una proprietà tipica della soluzione ottima di un *TSP*: gli archi che ne fanno parte non si incrociano mai fra loro.

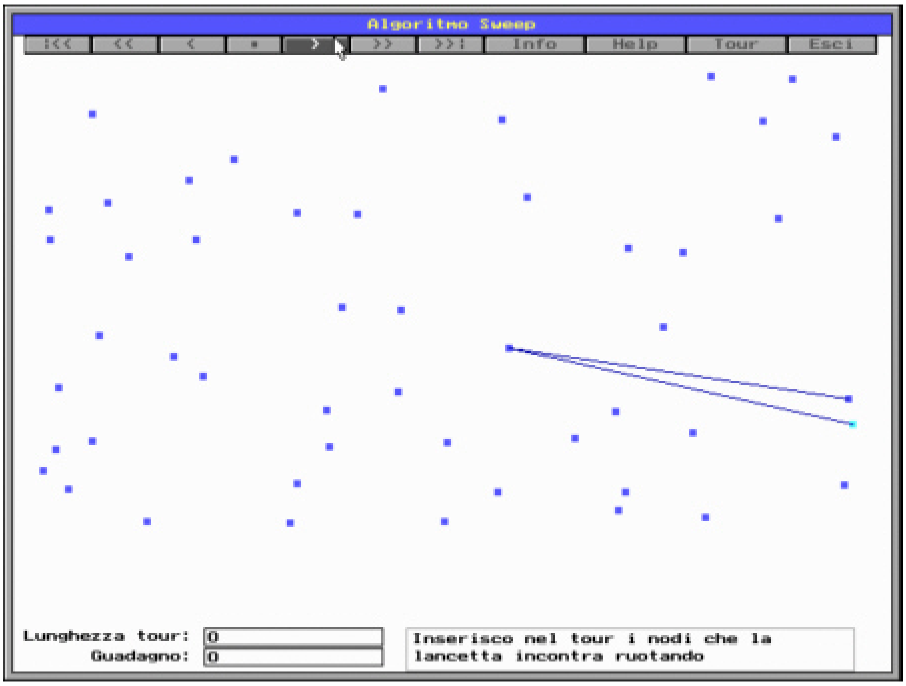
Il motivo per cui nessuna soluzione ottima di un *TSP* ha archi che si incrociano è che, nel caso ciò avvenisse, sostituendo i due archi che si incrociano con altri due archi che collegano gli stessi nodi si otterrebbe una soluzione migliore, dato che due lati opposti di un quadrilatero hanno sempre somma inferiore alle due diagonali del quadrilatero stesso.

La figura seguente illustra le prime fasi dell'algoritmo: il raggio comincia a scorrere l'area.



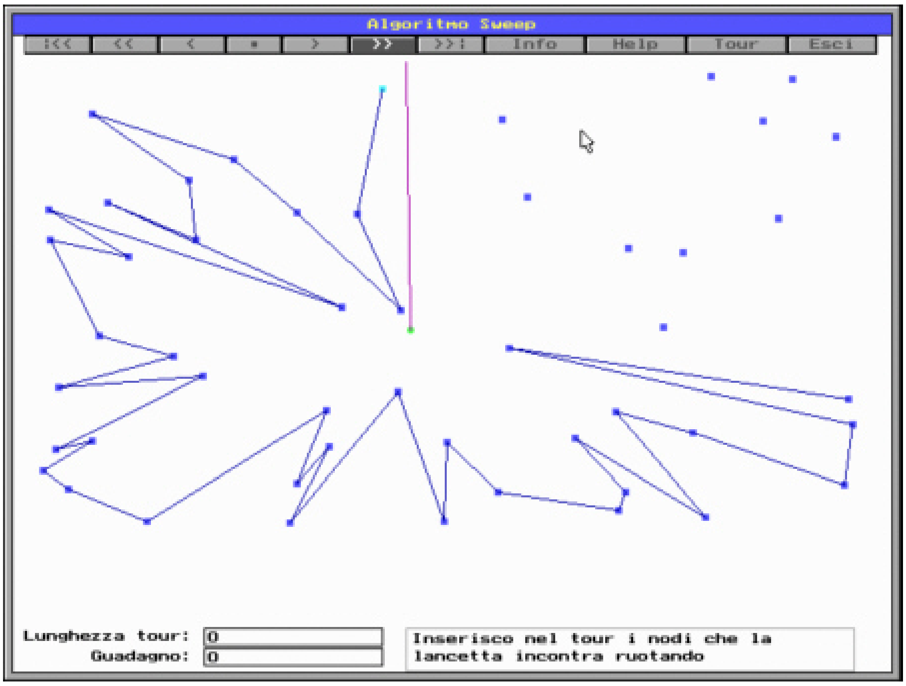
L'algoritmo "Sweep": il raggio (in viola) comincia a scorrere l'area

Nella figura successiva il raggio ha toccato tre nodi, ed essi sono stati inseriti, nello stesso ordine, nel percorso corrente. In azzurro è evidenziato l'ultimo nodo del percorso.



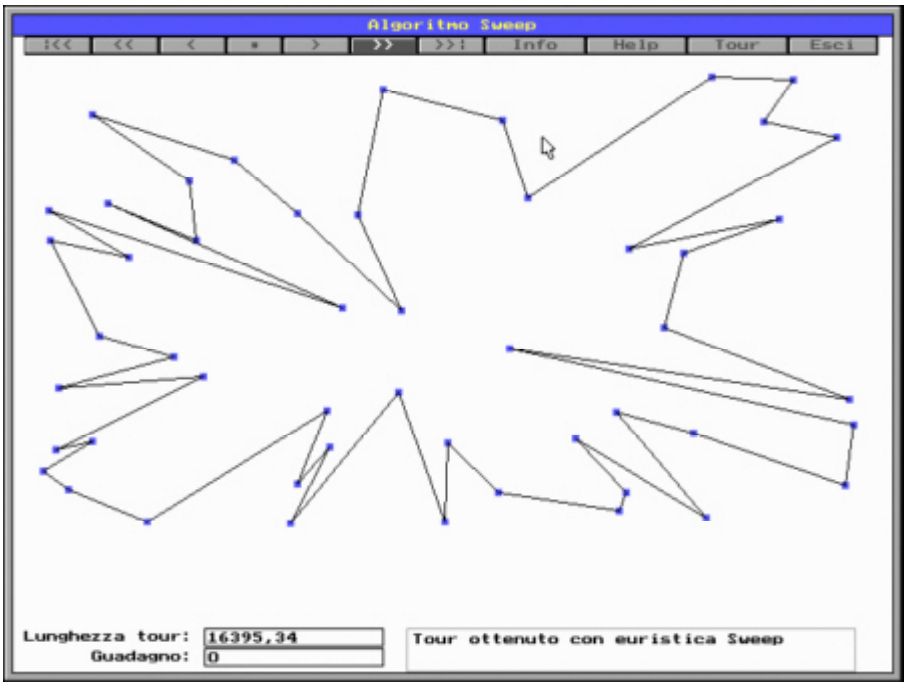
L'algoritmo "Sweep": : il percorso visita i primi tre nodi toccati dal raggio, nello stesso ordine in cui sono stati toccati

Il procedimento continua, generando un percorso che non torna mai su sé stesso, ma oscilla in continuazione fra nodi vicini al centro e nodi lontani (vedi figura seguente).



Una soluzione intermedia dell'algoritmo "Sweep"

Questo è il principale difetto dell'algoritmo *Sweep*, che non ottimizza in alcun modo le distanze percorse in direzione radiale. Infatti, la soluzione finale ha un andamento molto tortuoso e costa 16395,34.



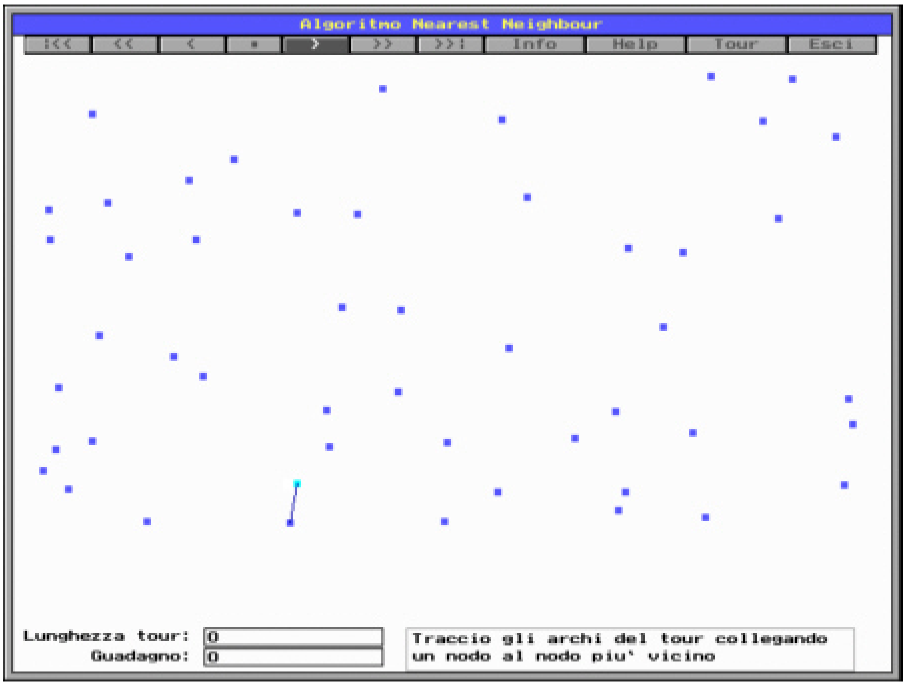
La soluzione finale dell'algoritmo "Sweep": si nota la continua oscillazione in direzione radiale

Top

Algoritmo *Nearest neighbour*

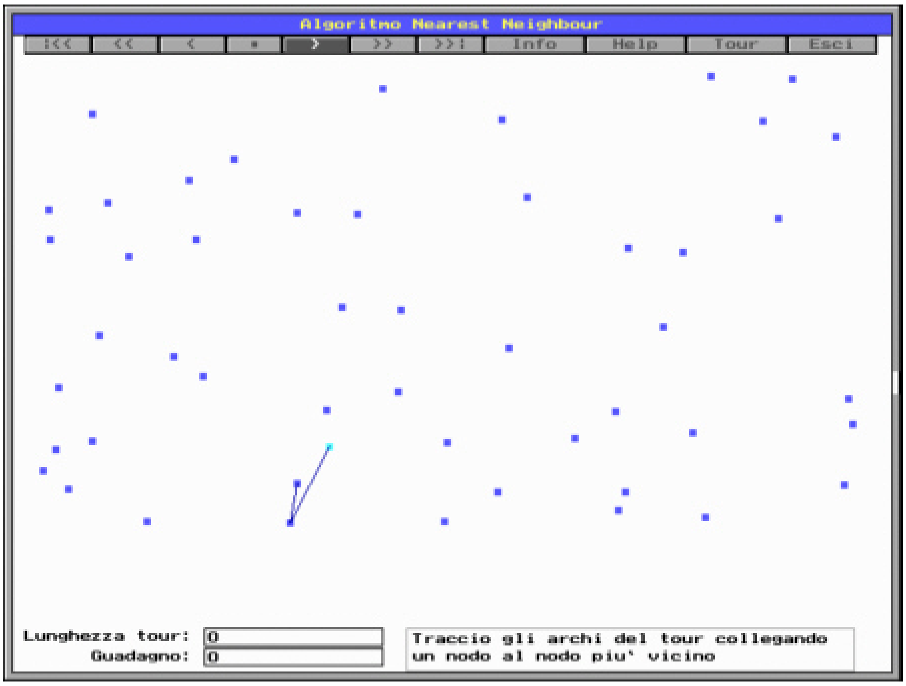
L'algoritmo *Nearest neighbour* si basa su un'idea molto semplice. Dato che l'obiettivo è trovare il ciclo di costo totale minimo, l'algoritmo parte da un nodo e comincia a percorrere l'arco di costo minimo che incide in quel nodo. In altre parole, raggiunge da esso il nodo più del grafo.

La figura seguente illustra questo primo passo: dal nodo iniziale, scelto arbitrariamente, si raggiunge il nodo più vicino, colorato in azzurro



Primo passo dell'algoritmo "Nearest neighbour": raggiungere il nodo più vicino a quello iniziale

I passi successivi (la figura seguente rappresenta il secondo) ripetono il primo: si cerca l'arco meno costoso che incida nell'ultimo nodo visitato e che non torni in nessuno dei precedenti. In altri termini, ad ogni passo si raggiunge il nodo non ancora visitato che sta più vicino all'ultimo nodo visitato.

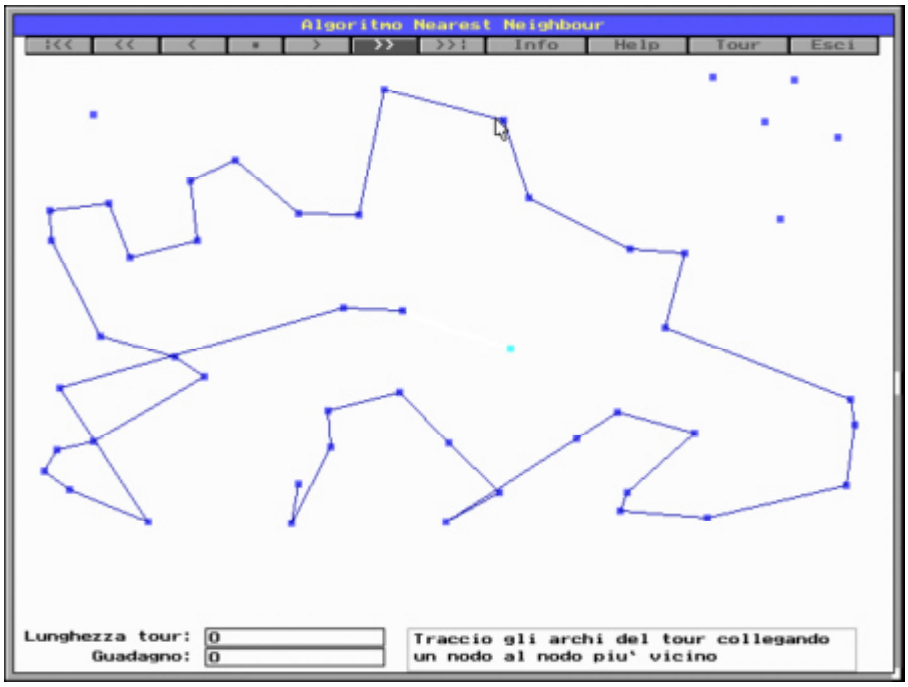


Secondo passo dell'algoritmo "Nearest neighbour": raggiungere il nodo più vicino al secondo nodo

L'idea dell'algoritmo è che facendo ogni volta la scelta più economica si possa ottenere la soluzione complessivamente più economica. Questa è l'idea di fondo dei così detti *algoritmi greedy* (dall'inglese, *ingordi*): dividere un compito in passi elementari ed eseguire ogni passo nel modo migliore.

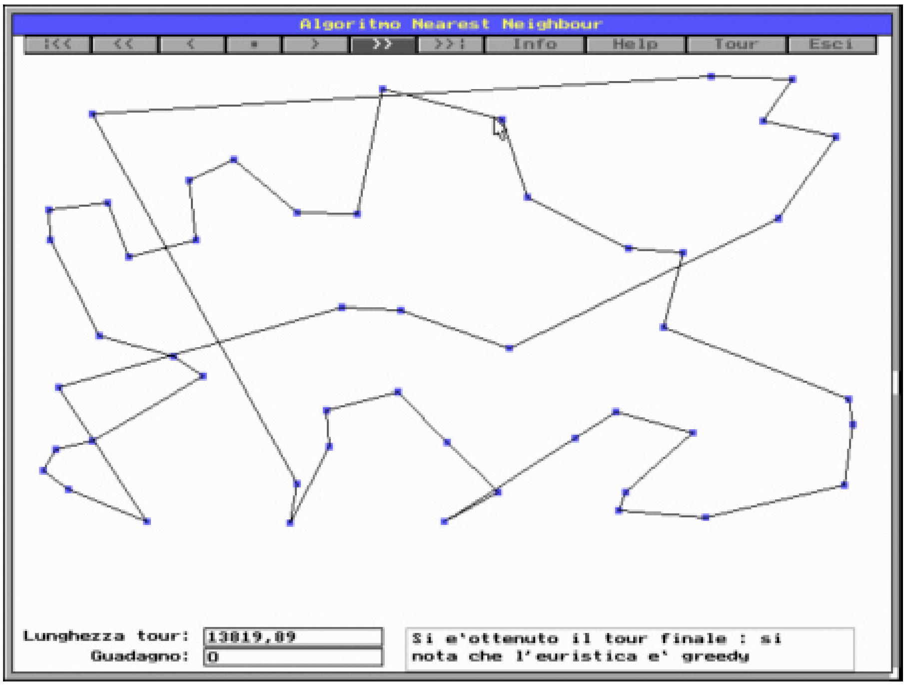
L'idea è logica e per alcuni problemi semplici funziona, cioè genera veramente una soluzione ottima. In generale, però, trascura un aspetto fondamentale: che ogni scelta elementare non solo contribuisce al costo complessivo (e quindi è bene fare scelte economiche), ma influisce sull'insieme delle scelte disponibili ai passi successivi. Questo significa che scelte molto economiche nei primi passi possono però rendere necessarie scelte molto costose nei passi successivi. Si parla spesso, al riguardo, della *miopia* degli algoritmi *greedy*, che sacrificano il bene futuro per un guadagno immediato.

Nel caso del *TSP*, dato che il percorso non può mai tornare sui propri passi, ad ogni passo il numero di scelte disponibili cala. La figura seguente mostra il classico caso in cui una scelta *greedy* prepara grossi svantaggi per il futuro: in alto a destra parecchi nodi non sono stati visitati nonostante che il commesso viaggiatore sia già passato vicino, perché nessuno di essi era il nodo più vicino in assoluto all'ultimo nodo del percorso. D'altra parte, prima del termine il commesso viaggiatore dovrà visitare anche quei nodi, spendendo molto più di quanto avrebbe potuto.



Una soluzione intermedia dell'algoritmo "Nearest neighbour": gia' si indovina la necessità futura di passi molto costosi

La figura seguente, infatti, illustra la soluzione finale ottenuta dall'algoritmo *Nearest neighbour*, che presenta nella parte terminale del percorso archi di costo molto elevato, per raggiungere i nodi tralasciati nei passi precedenti. Il suo costo totale è pari a 13819,89.



La soluzione finale dell'algoritmo "Nearest neighbour": si nota la scarsa qualita' della parte terminale del percorso

Algoritmo *Nearest insertion*

L'algoritmo *Nearest insertion* tenta di ovviare alla miopia dell'algoritmo *Nearest neighbour* concedendo un grado di libertà in più. Si procede ancora aggiungendo ogni volta a una soluzione parziale il nodo più vicino. Tuttavia, non è più obbligatorio aggiungere i nodi in un punto ben preciso della soluzione.

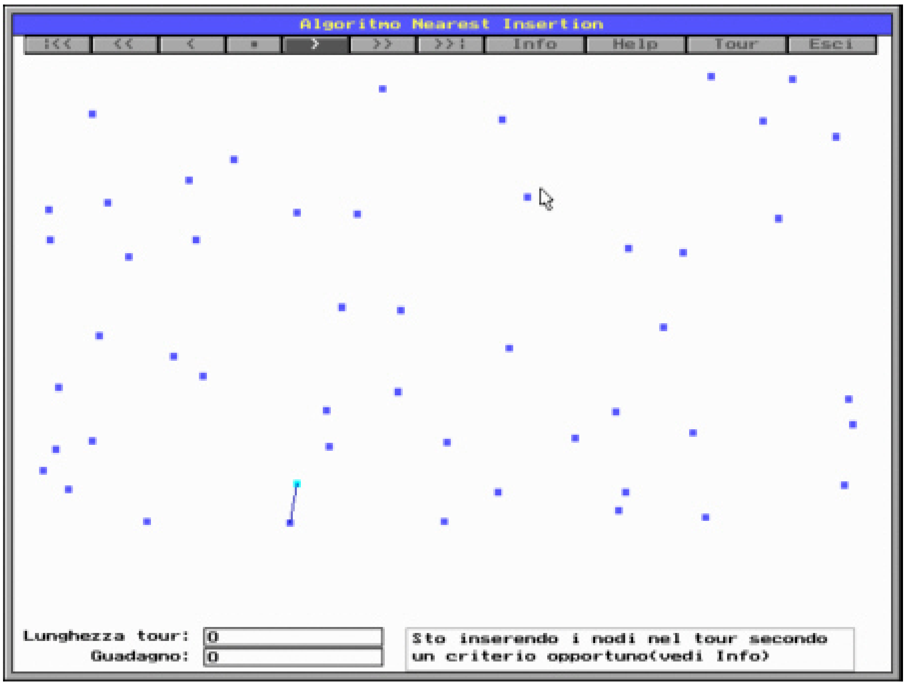
Al contrario, l'algoritmo:

1. conserva un ciclo, anziché un percorso

2. cerca il nodo più vicino a uno qualsiasi dei nodi del ciclo, anziché al solo nodo terminale del percorso

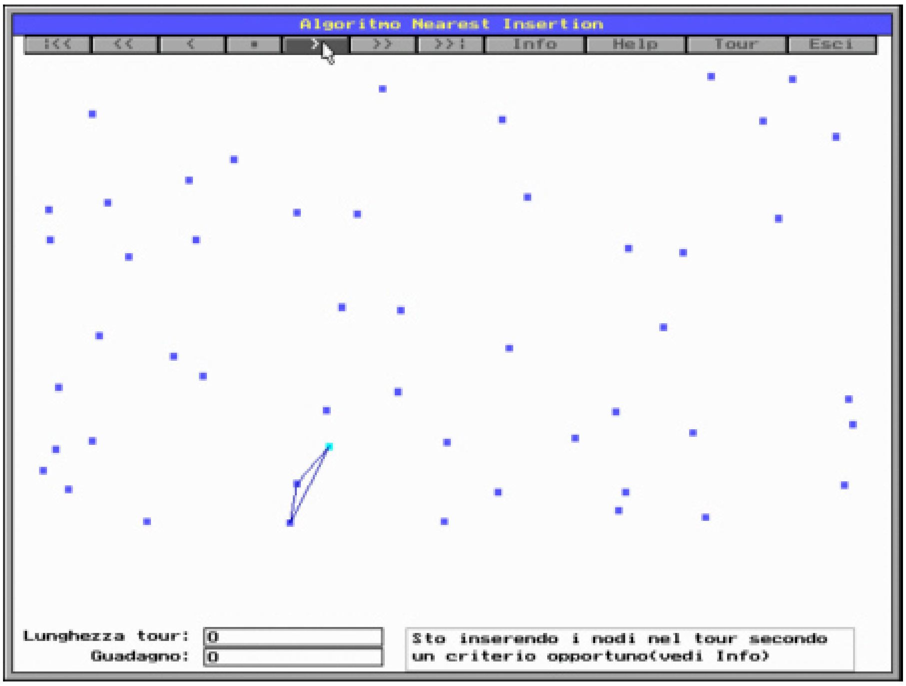
3. aggiunge il nuovo nodo nella posizione tale da dar luogo al ciclo più economico, anziché necessariamente al termine del percorso

La figura seguente illustra il primo passo, che in pratica coincide con quello dell'algoritmo precedente: dal nodo iniziale, scelto arbitrariamente, si raggiunge il nodo più vicino, dato che così facendo si genera il ciclo più corto.



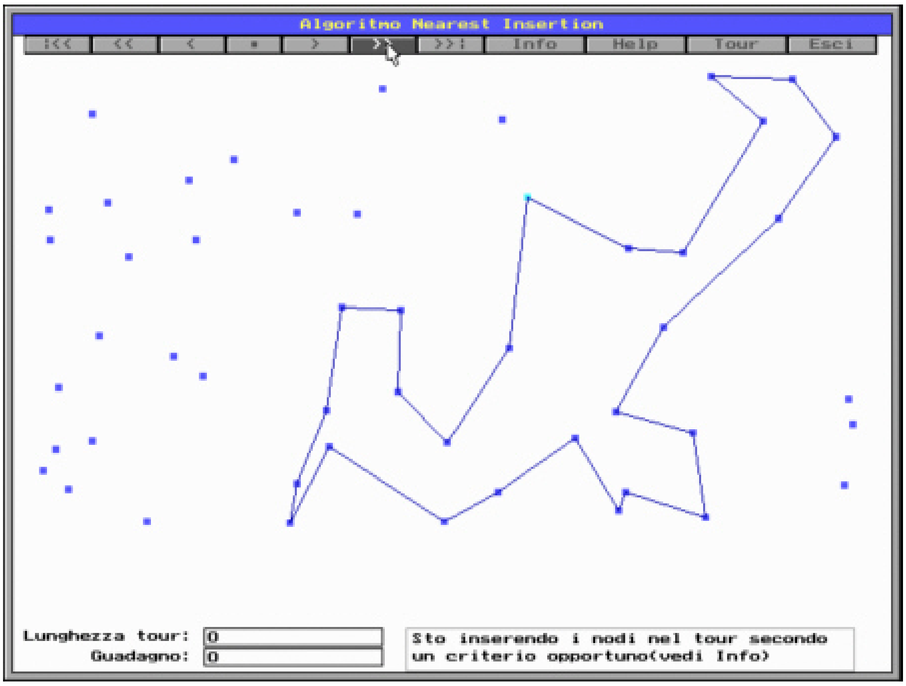
Primo passo dell'algoritmo "Nearest insertion": partendo dal nodo iniziale, si crea il ciclo piu' corto possibile

Al secondo passo, si crea un vero e proprio ciclo da tre nodi (vedi la figura seguente).



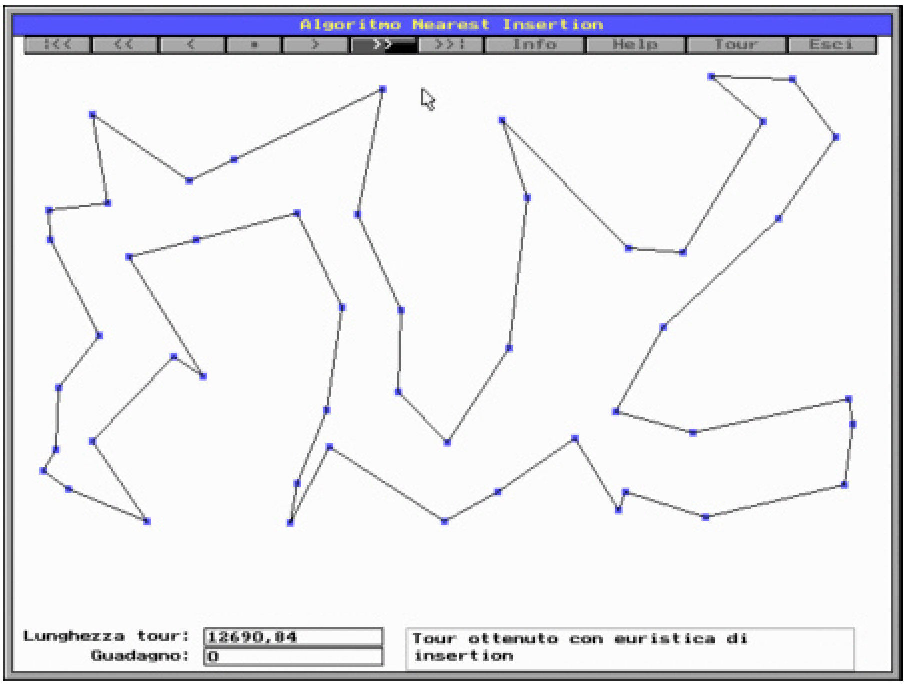
Secondo passo dell'algoritmo "Nearest insertion": si crea un ciclo da tre nodi

Nei passi successiv ,i appare ev identela maggior flessibilità dell'algoritmo: i nodi possono v enireaggiunti in qualsiasi posiz ionedel ciclo. Quindi, anche se rimangono non v isitatisino agli ultimi passi, lo saranno alla fine in modo non eccessiv amente costoso.



Una soluzione intermedia dell'algoritmo 'Nearest insertion'

La figura seguente illustra la soluzione finale, che non presenta archi di costo molto alto e ha un costo totale pari a 12690.84.



La soluzione finale dell'algoritmo "Nearest insertion"

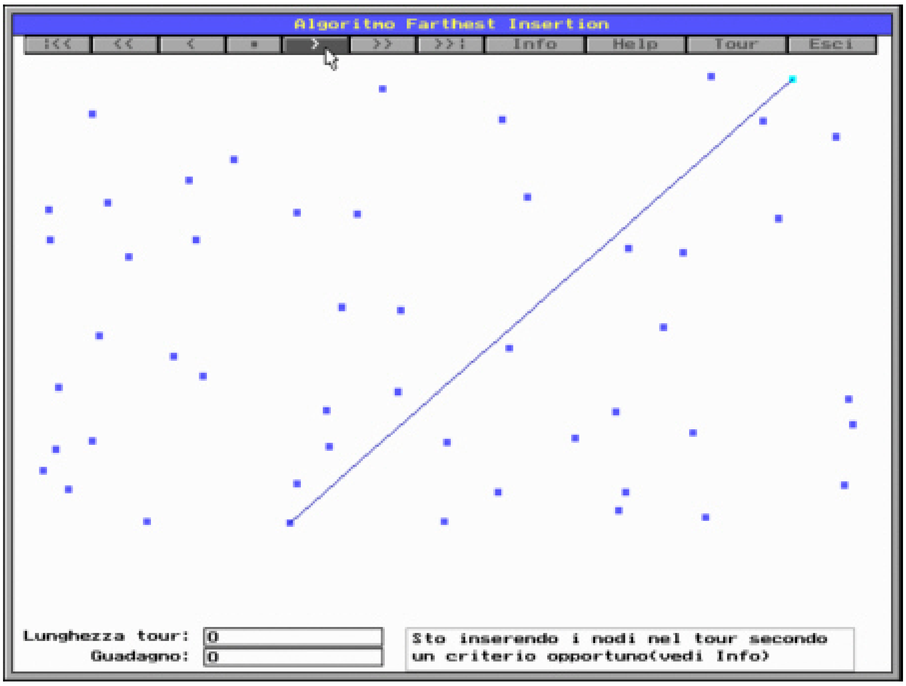
Tuttavia, la forma della soluzione appare troppo contorta per essere davvero buona. Il punto è che anche l'algoritmo *Nearest insertion*, pur senza commettere errori macroscopici come il precedente, è miope, dato che tende a seguire i percorsi migliori solo per raggiungere i nodi vicini, senza preoccuparsi, sino agli ultimi passi, dei nodi lontani. Questi vengono alla fine serviti nel modo migliore, ma si tratta del modo migliore consentito dalla forma che il ciclo ha ormai assunto nelle sue linee generali.

Algoritmo *Farthest insertion*

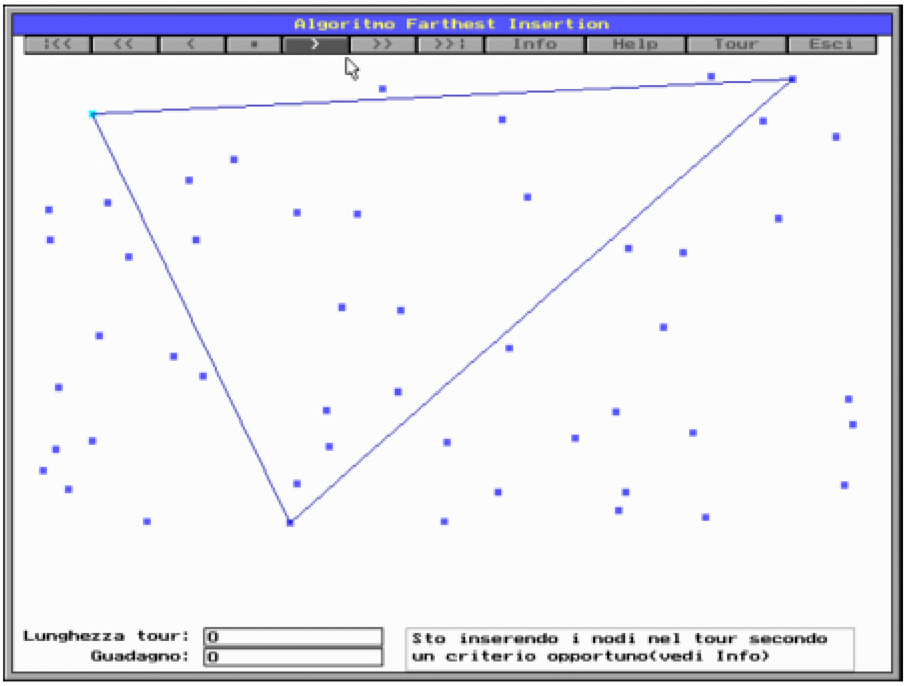
L'algoritmo *Farthest insertion* tiene conto in partenza del fatto che tutti i nodi vanno visitati e si preoccupa anzitutto di assicurare una visita economica ai nodi più difficili, cioè a quelli più lontani.

In altre parole, a ogni passo l'algoritmo trova il nodo più lontano dal ciclo corrente e lo inserisce nel modo migliore.

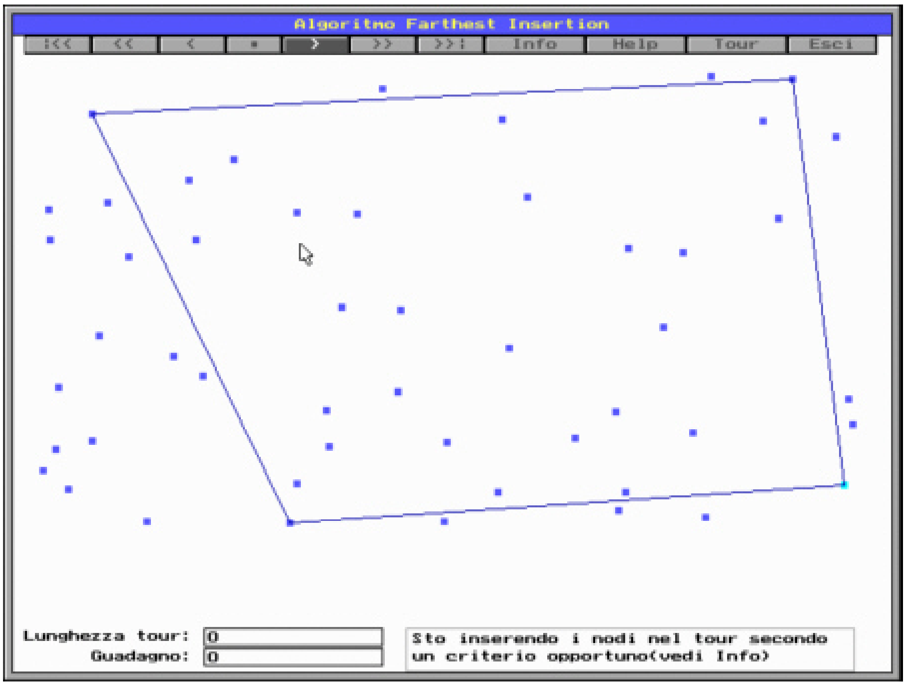
Il concetto è un po' controintuitivo, ma bastano le schermate dei primi passi dell'algoritmo (vedi le figure seguenti) per intuire che la forma del ciclo risultante è molto più regolare e intelligente.



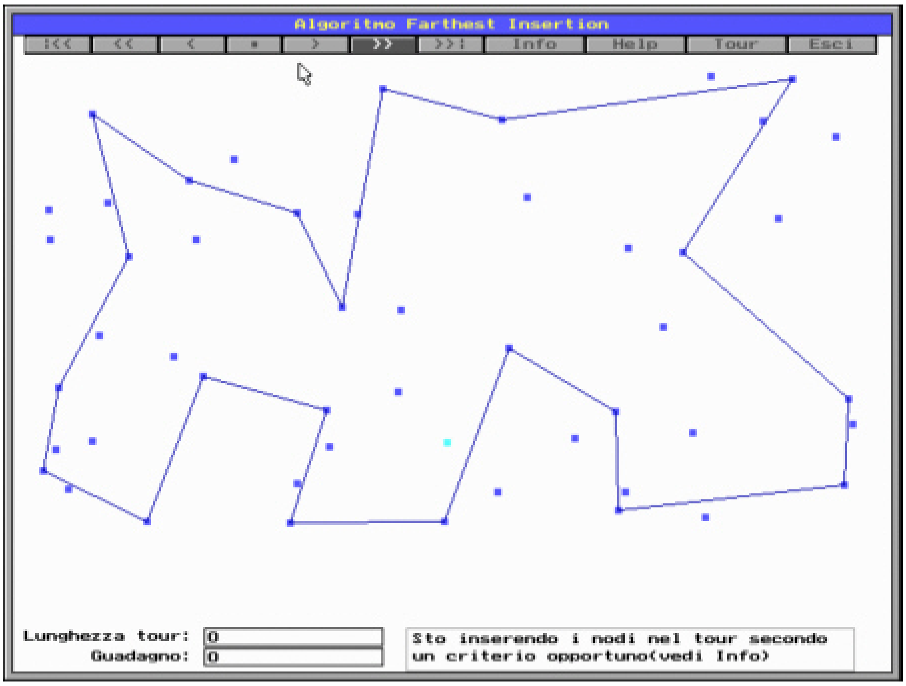
Primo passo dell'algoritmo "Farthest insertion": partendo dal nodo iniziale, si raggiunge il nodo piu' lontano



Secondo passo dell'algoritmo "Farthest insertion": si costruisce il ciclo piu' corto che raggiunga il nodo più lontano dal ciclo precedente

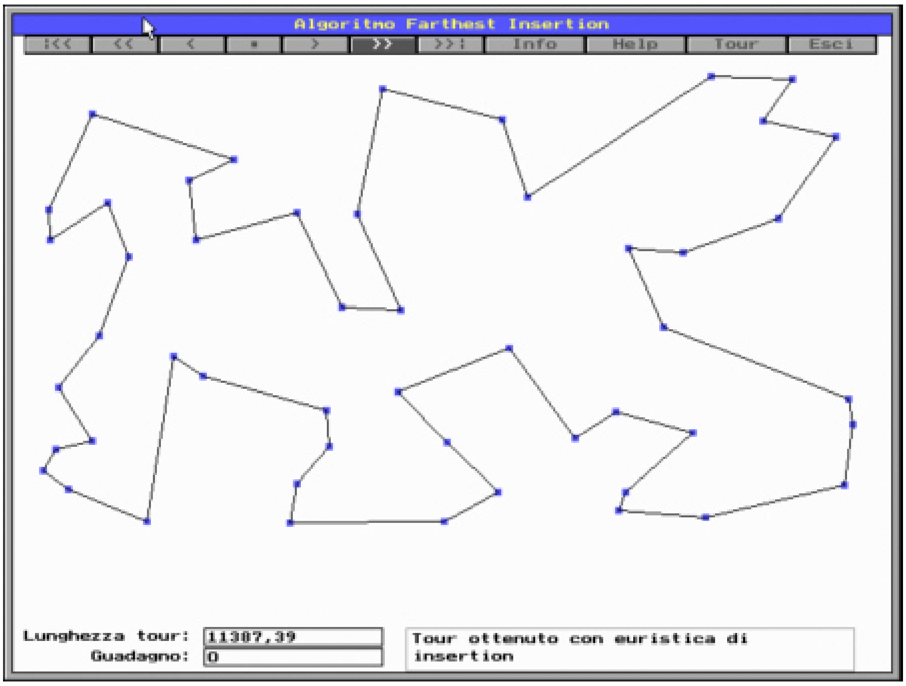


Terzo passo dell'algoritmo "Farthest insertion": si costruisce il ciclo piu' corto che raggiunga il nodo più lontano dal ciclo precedente



Una soluzione intermedia dell'algoritmo "Farthest insertion"

La figura seguente illustra la soluzione finale, che presenta una forma molto regolare e ha un costo totale pari a 11387.89.



La soluzione finale dell'algoritmo "Farthest insertion"

Top

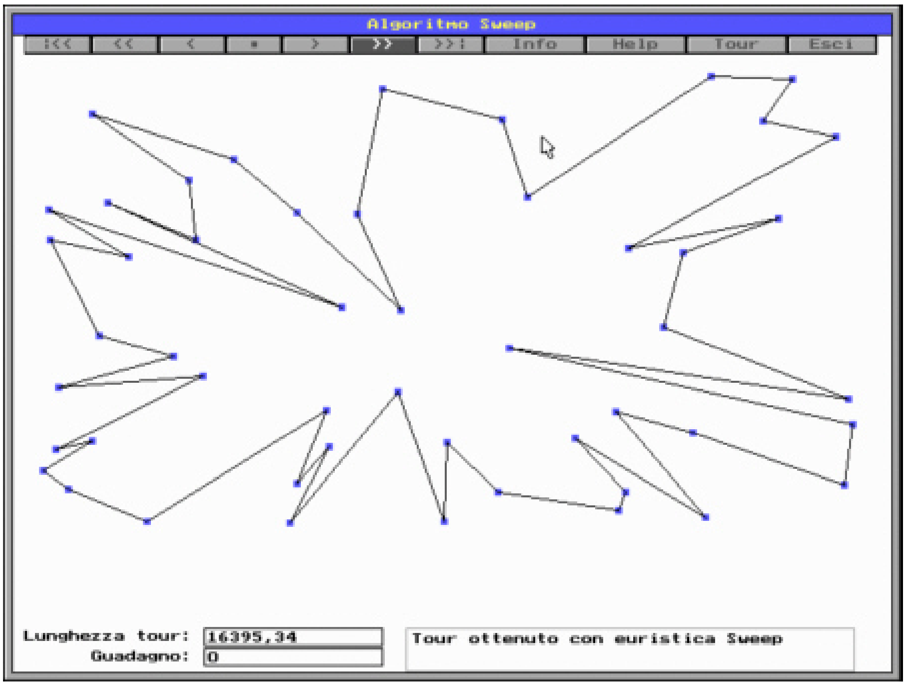
Gli algoritmi di ricerca locale classica

Gli algoritmi di ricerca locale si basano sul concetto di *intorno* di una soluzione.

L'intorno di una soluzione è un insieme di altre soluzioni che si possono ottenere da essa con un'opportuna famiglia di trasformazioni. Ogni trasformazione della famiglia genera una diversa soluzione dell'intorno. Si parla di intorno perché in generale le soluzioni trasformate sono simili a quella di partenza.

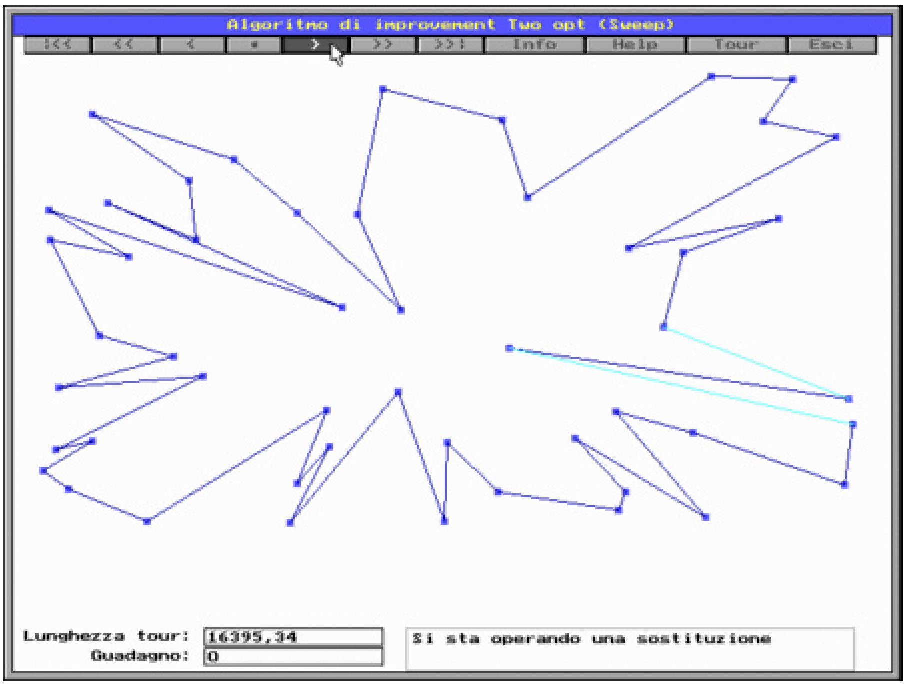
Gli algoritmi di ricerca locale partono da una soluzione ottenuta in qualche modo (ad esempio con un algoritmo costruttivo). Quindi, generano le soluzioni dell'intorno, cercandone una migliore di quella di partenza. Fra quelle migliori, ne scelgono da sostituire alla soluzione iniziale e ripartono da capo, generando ed esplorando l'intorno di quest'ultima. A seconda degli algoritmi, si sceglie la prima soluzione migliorante che si trova, oppure si generano tutte e poi si sceglie la migliore, o si seguono altre strategie ancora.

Prendiamo ad esempio l'algoritmo *Two-opt*, ovvero dei *2-scambi*. Esso assume come famiglia di trasformazioni per generare l'intorno quella che elimina due archi della soluzione di partenza e li sostituisce con altri due in modo da ricostituire un ciclo unico. La figura seguente rappresenta la soluzione di partenza che consideriamo per il consueto *TSP* di esempio: è la soluzione generata dall'algoritmo *Sweep*.



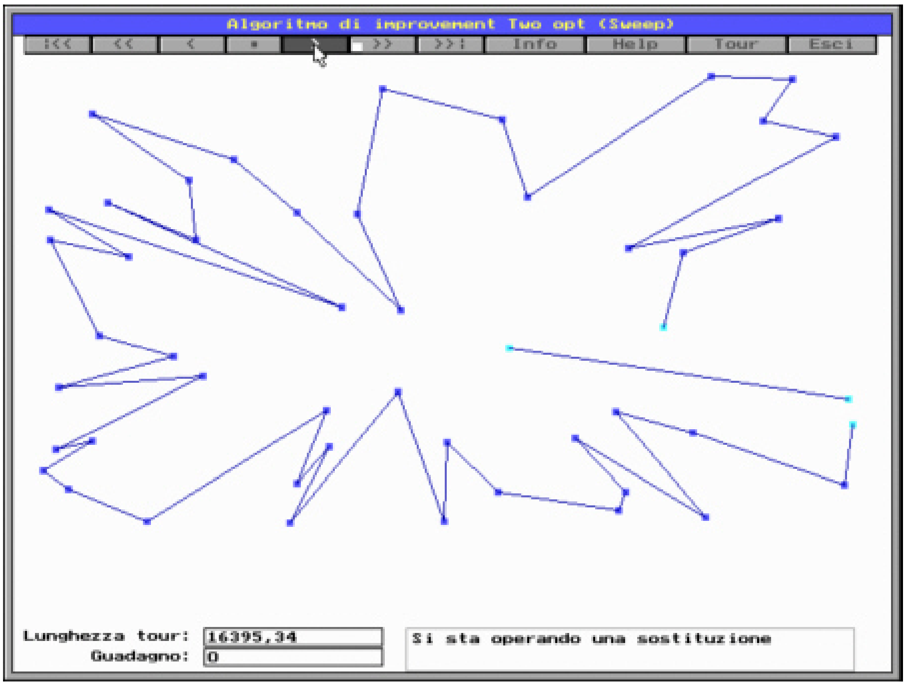
La soluzione iniziale dell'algoritmo di ricerca locale "Two-opt"

L'algoritmo *Two-opt* individua due archi nella soluzione, indicati in azzurro nella figura seguente.

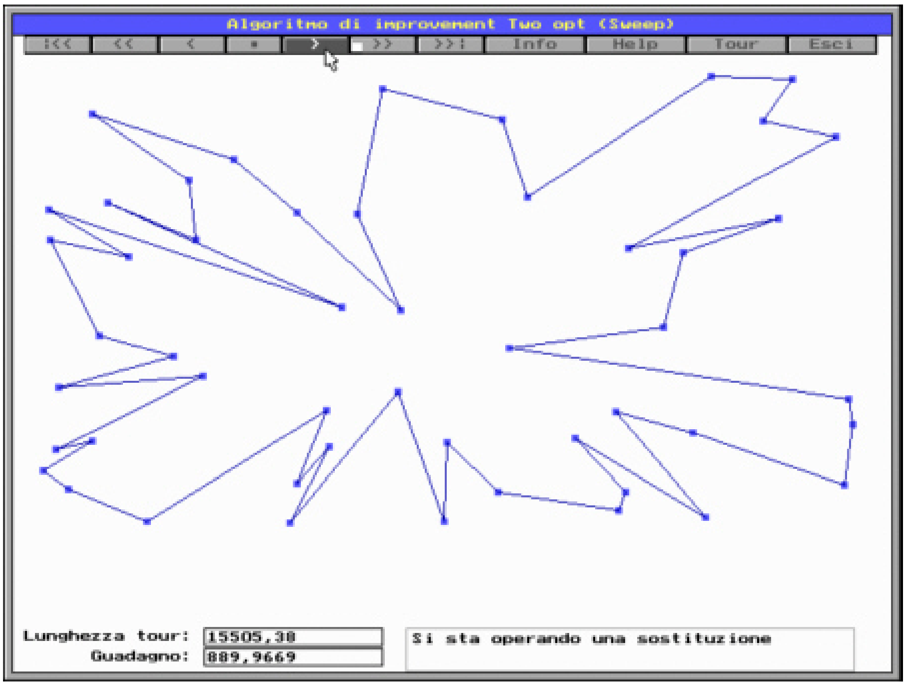


L'algoritmo "Two-opt" sceglie due archi della soluzione iniziale

Quindi, li elimina e li sostituisce con altri due archi nell'unico modo possibile per mantenere un solo ciclo che passi per tutti i nodi, cioè una soluzione ammissibile al *TSP*.



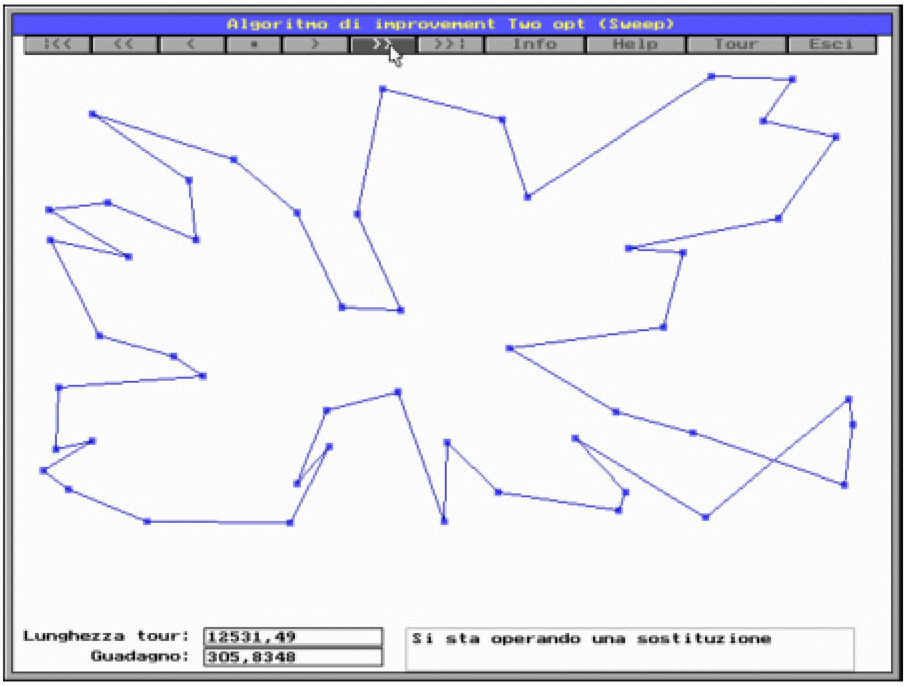
L'algoritmo "Two-opt" elimina i due archi



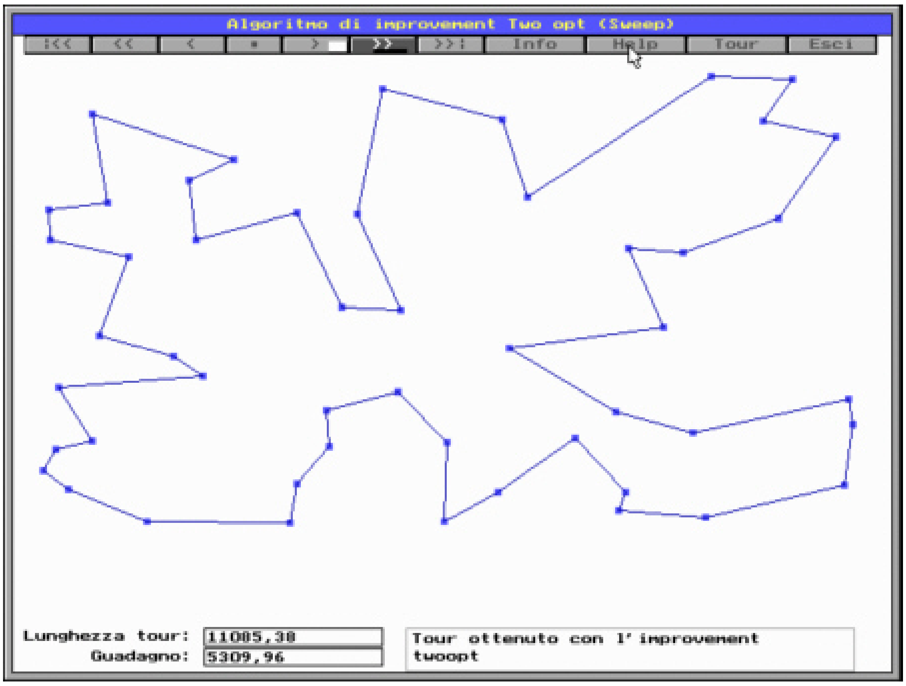
L'algoritmo "Two-opt" aggiunge due nuovi archi

È evidente che la nuova soluzione è molto meno costosa della precedente. In effetti, l'algoritmo ha seguito la strategia di scegliere la miglior soluzione dell'intorno, cioè di eseguire, fra tutti gli scambi di archi possibili, quello che porta il massimo risparmio.

Di passo in passo, la soluzione va migliorando. Si noti che è possibile, in passi intermedi, che la soluzione corrente presenti degli incroci di archi (come nella figura seguente), ma essi verranno sicuramente eliminati da passi successivi. Infatti, il motivo per cui nella soluzione ottima non vi sono incroci è proprio che in caso contrario sarebbe possibile scambiare coppie di archi per ottenere un risparmio.



Una soluzione intermedia dell'algoritmo "Two-opt"



La soluzione finale dell'algoritmo "Two-opt"

La soluzione finale dell'algoritmo, pur partendo da un ciclo molto tortuoso, è decisamente regolare e costa soltanto 11085,38.

Top

Punti deboli degli algoritmi di ricerca locale classica

Il cuore degli algoritmi di ricerca locale è la definizione dell'intorno. Questo deve essere ampio, perché altrimenti non vi si trovano buone soluzioni, ma deve essere ristretto perché altrimenti esplorarlo in modo esaustivo diventa troppo pesante.

Il difetto fondamentale di questi algoritmi sta nel fatto che non trovano l'*ottimo globale*, cioè la soluzione migliore fra tutte quelle ammissibili, bensì l'*ottimo locale*, cioè la soluzione migliore fra tutte quelle dell'intorno.

Esistono molti ottimi locali, in genere, e quello che viene generato dall'algoritmo dipende molto dalla soluzione di partenza che gli si fornisce. D'altra parte, non si sono regole generali per fornire una buona soluzione di partenza. Di solito, migliore è la soluzione di partenza, migliore è l'ottimo locale generato, ma proprio l'algoritmo *Sweep* è un controesempio, dato che produce cattive soluzioni, ma, seguito da un algoritmo di ricerca locale, dà spesso soluzioni molto buone.

Un modo parziale di sfuggire alla dipendenza dalla soluzione iniziale è quello di ripetere l'esecuzione con molte diverse soluzioni di partenza. Si parla allora di *restart* o *multistart*.

Gli algoritmi di ricerca locale classica, una volta giunti in un ottimo locale, si fermano. Benché fuori dell'intorno possano esistere soluzioni migliori, magari di molto, tali algoritmi sono incapaci di trovarle, perché non "vedono" nulla fuori dell'intorno. Questa è la *miopia* degli algoritmi di ricerca locale, che fa da contraltare a quella degli algoritmi costruttivi.

Le metaeuristiche di ricerca locale

Le metaeuristiche di ricerca locale nascono per ovviare al problema degli ottimi locali. Nel momento in cui un algoritmo di ricerca locale classico giunge in un ottimo locale, si ferma. Occorre invece proseguire la ricerca altrove, nel tentativo di trovare altri ottimi locali, e possibilmente un ottimo globale.

L'idea di proseguire la ricerca si scontra però con una difficoltà pratica. Nella ricerca locale classica la regola di passaggio da una soluzione alla successiva è semplicissima: migliorare sempre. Nelle metaeuristiche, è ammesso peggiorare, ma la tendenza è sempre quella al miglioramento. Questo significa che, appena abbandonato un ottimo locale, subito l'algoritmo tende a ritornarvi, per migliorare la soluzione corrente. Descriviamo per sommi capi due modi di superare questa difficoltà.

Simulated Annealing

Si tratta di rendere la ricerca parzialmente casuale. Nell'esplorare l'intorno della soluzione corrente, se si incontra una soluzione migliorante, la si accetta e sostituisce immediatamente a quella data. Se invece si incontra una soluzione peggiorante, si decide in modo casuale se accettare o no la nuova soluzione. La probabilità di accettarla è tanto più forte quanto meno peggiorante è la soluzione.

Tale probabilità dipende poi da un parametro, detto *temperatura* che per valori alti rende molto probabile accettare soluzioni peggioranti, mentre per valori bassi lo rende poco probabile. La temperatura va calando nel corso dell'esecuzione dell'algoritmo, in modo che, via via che diventa sempre più difficile trovare soluzioni migliori di quelle già visitate, diventa anche sempre più difficile accettare peggioramenti. Così la ricerca ha un termine.

Tabu Search

Si tratta di dotare la ricerca di memoria. Passo per passo, si sostituisce alla soluzione corrente la migliore soluzione dell'intorno, vale a dire una soluzione migliorante, oppure quella che meno peggiora il valore dell'obiettivo.

Per evitare di ricadere in ottimi locali già visitati, si conserva memoria in una *lista tabù* di alcune loro caratteristiche distintive (il fatto di contenere certi archi, ad esempio). Nell'esplorare l'intorno si rifiutano tutte le soluzioni che hanno quelle caratteristiche.

Poiché questo vieta non solo le soluzioni già visitate, ma anche altre soluzioni che semplicemente somigliano loro, per evitare di restringere eccessivamente la ricerca, il divieto viene reso temporaneo. Dura cioè solo un certo numero di passi, che si considera tale da far allontanare la soluzione corrente dall'ottimo locale abbastanza per non ricadervi.

Il *Tabu Search*, con una serie di modifiche che ne accrescono l'efficacia e la capacità di esplorare l'insieme delle soluzioni, è oggi forse la tecnica euristica più potente.

Gli algoritmi genetici

Gli algoritmi genetici si basano su un'analogia con la genetica delle popolazioni. Non operano su una soluzione sola, che viene via via costruita o modificata, ma su un'intera popolazione di soluzioni, generate in qualche modo (a caso o con euristiche costruttive), che evolvono nel tempo.

Le soluzioni sono codificate come sequenze di simboli (spesso 0 e 1, ma a volte si usano alfabeti più complessi) dette *cromosomi*. A ogni passo, le soluzioni si riproducono, oppure no, con una probabilitè che dipende dal costo della soluzione stessa. È definita infatti una funzione di *fitness* che traduce il costo in una probabilitè di riprodursi, in modo che le soluzioni più economiche abbiano probabilità maggiori.

In generale, la riproduzione non è meccanica, dato che:

- i cromosomi subiscono casuali modifiche locali attraverso un operatore di *mutazione*
- i cromosomi si combinano a coppie, scambiandosi porzioni della propria sequenza di simboli attraverso un operatore di *cross-over*

Generazione dopo generazione, le soluzioni più costose tendono a scomparire perché si riproducono meno spesso, mentre le soluzioni migliori si combinano, generando talora soluzioni che presentano il meglio di entrambi i genitori.

L'aspetto casuale del processo e la presenza di un'intera popolazione di soluzioni consente anche a soluzioni di scarsa qualità, che però contengano qualche caratteristica utile, di farla fruttare, cedendola ad altre attraverso la ricombinazione.

Inoltre, l'uso di una popolazione di soluzioni permette di esplorare al tempo stesso diverse regioni dell'insieme delle popolazioni, riducendo la dipendenza dalla popolazione iniziale. D'altra parte, gli algoritmi genetici sono lenti e incontrollabili per la loro casualità.

L'*Ant System*

L'*Ant System* nasce da un'analogia con il comportamento delle colonie di formiche nella costruzione dei percorsi di foraggiamento. L'idea è disporre di una popolazione di *agenti*, che costruiscono simultaneamente soluzioni indipendenti a un problema.

Per fare ciò, impiegano qualche semplice metodo, generalmente costruttivo e *greedy* nella sua concezione, che comporta una sequenza di scelte elementari. Queste scelte sono eseguite in parte in modo casuale, preferendo non l'alternativa rigorosamente più economica, ma assegnando alle alternative più economiche probabilità via via sempre crescenti.

I risultati non saranno generalmente di buona qualità, ma permettono di farsi un'idea di quali scelte elementari (l'uso di un arco ben determinato, ad esempio) abbiano portato alle conseguenze più desiderabili. Il metodo procede quindi a premiare queste scelte associando ad ognuna un valore di *traccia* tanto più alto quanto più buone sono le soluzioni che hanno generato.

A questo punto, una nuova generazione di agenti procede a costruire una seconda popolazione di soluzioni, con lo stesso semplice algoritmo costruttivo, e compiendo le scelte elementari con una probabilità che dipende sia dal costo della singola scelta sia dalla traccia che vi è associata. In questo modo, l'algoritmo tiene conto non solo del costo della singola scelta, ma anche del suo effetto globale sulla soluzione.

Il procedimento si ripete più e più volte, in maniera da concentrare la ricerca sulle soluzioni più promettenti. D'altra parte, per evitare che le prime soluzioni trovate, che sono in genere poco interessanti, influenzino stabilmente la ricerca nei passi successivi, le tracce vengono gradualmente indebolite nel tempo, così che rimangano solo quelle che vengono continuamente rinforzate perché generano continuamente le soluzioni migliori.

Torna al sommario


BACK


NEXT